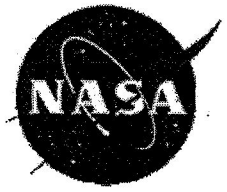


A Core Plug and Play Architecture for Reusable Flight Software Systems

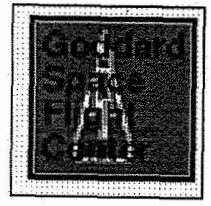
Jonathan Wilmot
Jonathan.J.Wilmot@nasa.gov
NASA GSFC Code 582

Mission Challenges

- Missions pushing for more spacecraft autonomy
- Increasing mission complexity
- Reduce cost, schedule, and risk
- Maintain flight software quality

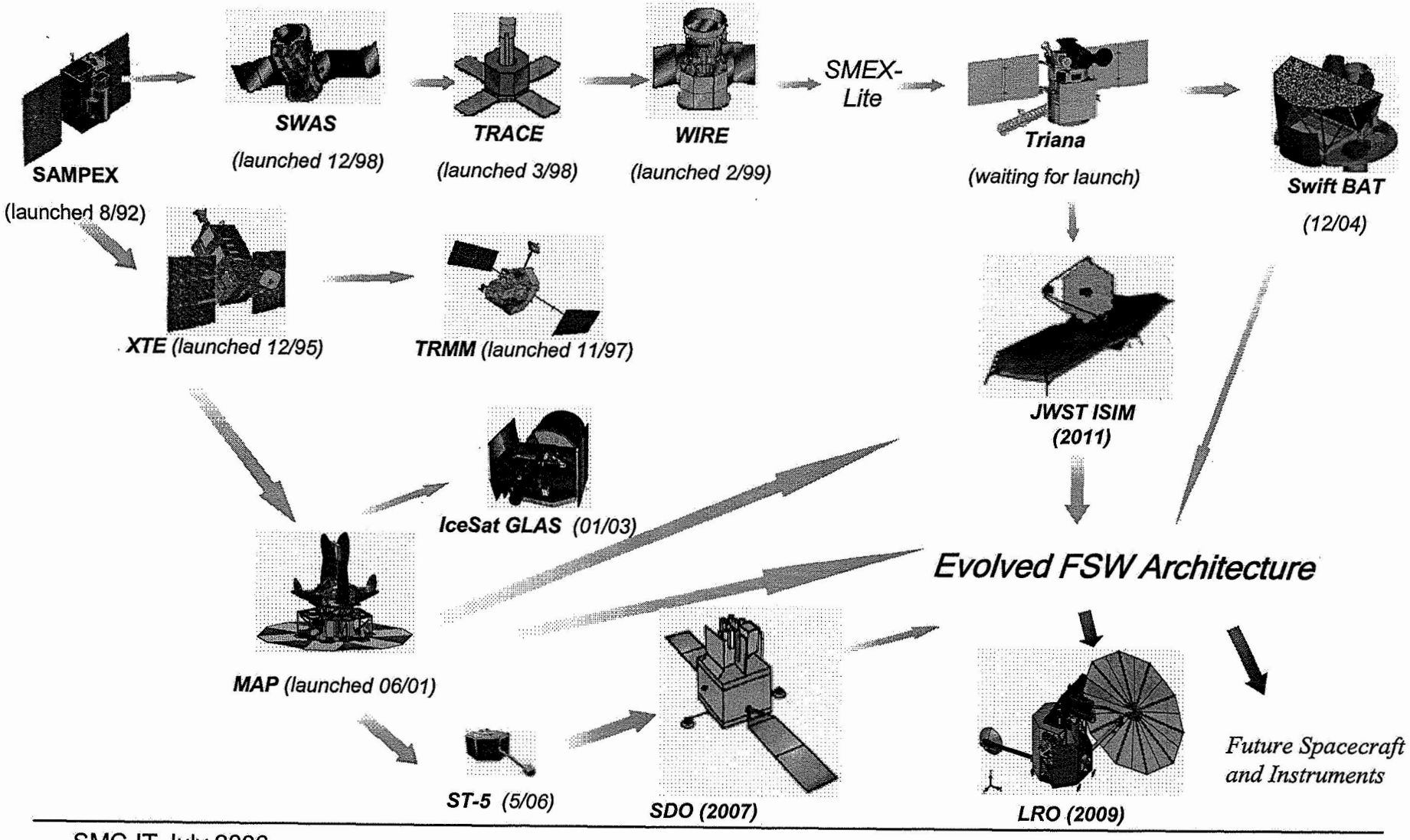
- GSFC approach
  - Platform-independent, service oriented flight software framework
  - "Configure and Play" catalog of reusable components

Heritage Analysis

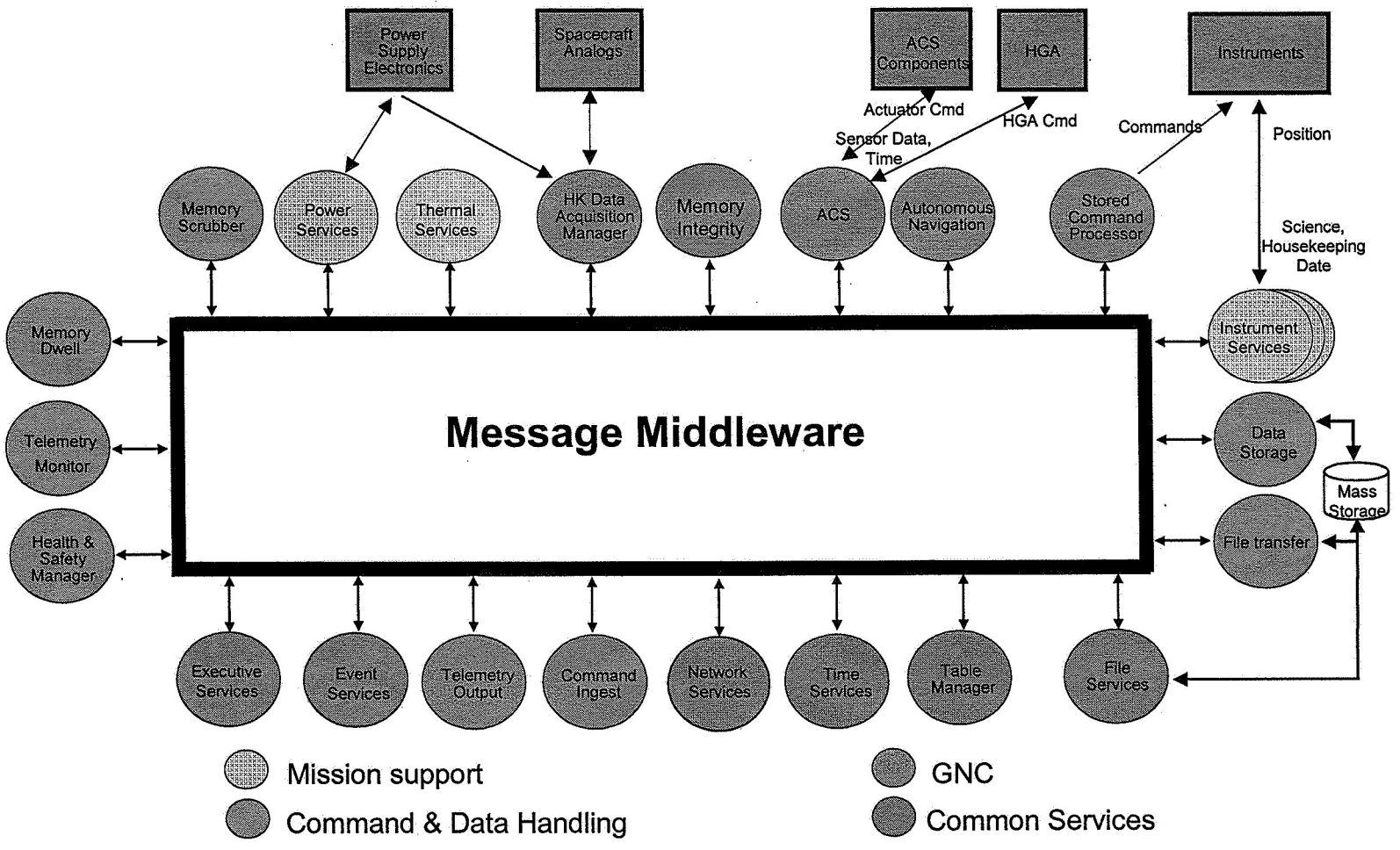


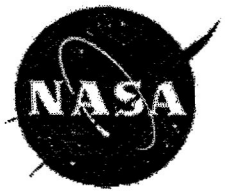
- Bring together senior software engineers with wide experiences
- Analyze previous missions architectures
- Extract common requirements
- Establish architecture goals for next generation with evolvability and expandability in mind
- Develop formal requirements

Heritage Mission Analysis

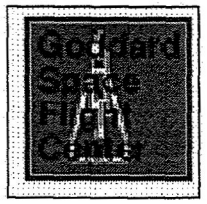


Heritage Architecture

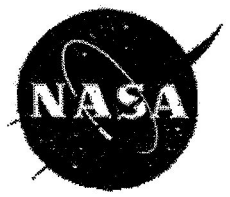




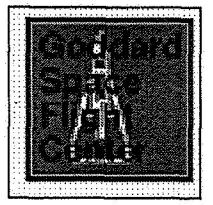
Performance Design Goals



- Spaceflight platforms are performance constrained
  - Small core software footprint
    - 512 kilobytes to 1 megabyte
  - Scaleable core memory utilization
    - 1 to 2 megabytes
  - Core processor utilization
    - Less than 5% over 1 second



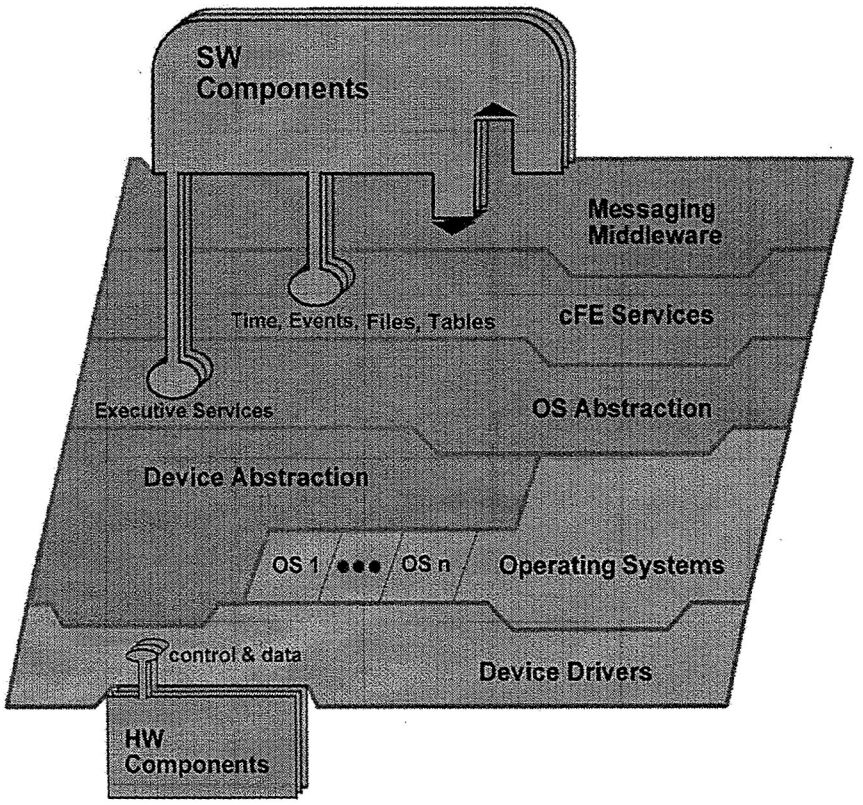
Run-Time Environment



- Component Modularity
  - Software decomposed along clean functional lines
  - Components interact with system through API
  - Components can be individually compiled and linked
- Dynamic Loading/Unloading
  - Components can be loaded and unloaded from running system
  - Support both static and dynamic linking
  - Component cleanup
- Operational Transparency
  - Operator visibility into the current running state
  - System utilization monitors
  - Built in diagnostics

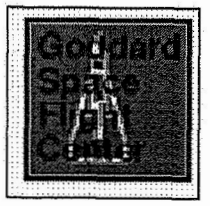
Implementation

- Layered Architecture
- Publish/Subscribe Middleware/Bus
- Standard Application Programmer Interface
- Run-time Services

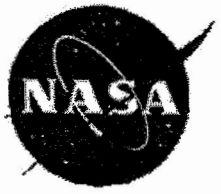


Core Flight Executive (cFE)

cFE Services



- OS Abstraction
  - Normalized API and services for common flight operating systems
- Executive Services
  - System startup/restart
  - Component start/stop/load/unload
  - Exception and interrupt handling
  - System logs
  - Performance monitoring utilities



cFE Services

- Software Bus
  - Publish and subscribe messaging middleware
  - Local and networked inter-component messaging
  - One-to-one, one-to-many, many-to-one
  - Poll, Pend, and Quality of Service
- Event Handler
  - Event formatting and filtering
  - Local logging
  - Event port selection

cFE Services

- Table Management
  - Named data storage
  - Load/Dump/Activate
- Time Management
  - Time maintenance and external interfaces
  - Local distribution
  - Time utilities

Software Component Example

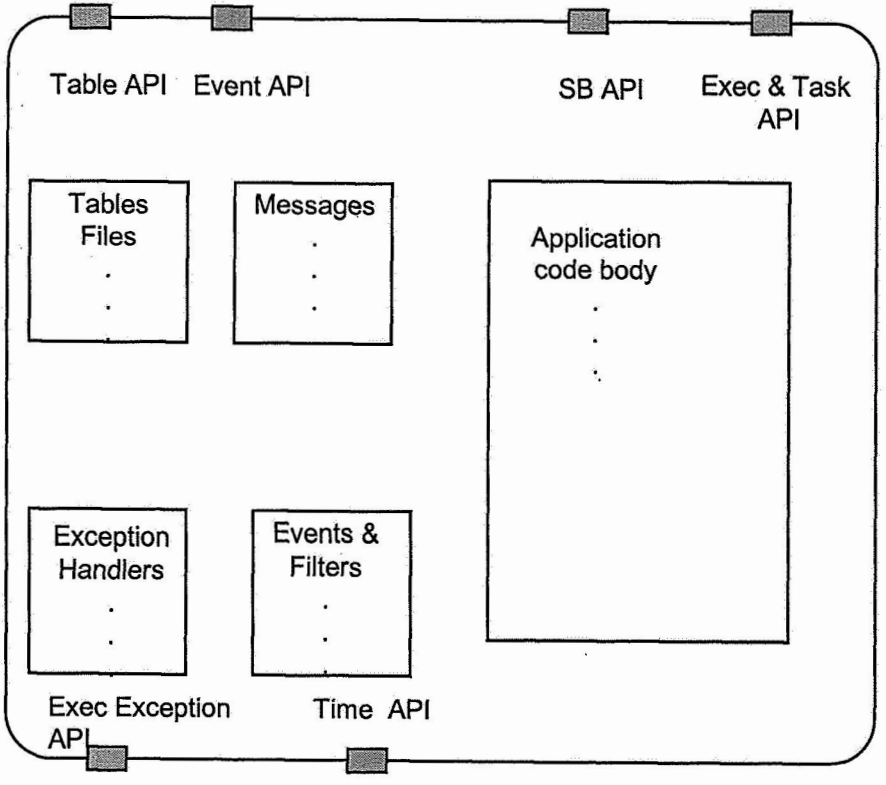
- Interface only through five cFE API's.

- cFE services track component resources
  - Capability to clean up after component faults
  - Provides utilization statistics

- A components contains all data needed to define it's operation.

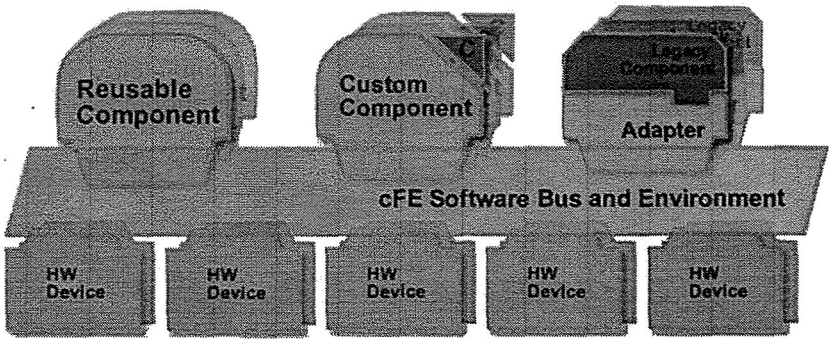
- Components register for services
  - Register exception handlers
  - Register Event counters and filter
  - Register Tables
  - Publish messages
  - Subscribe to messages

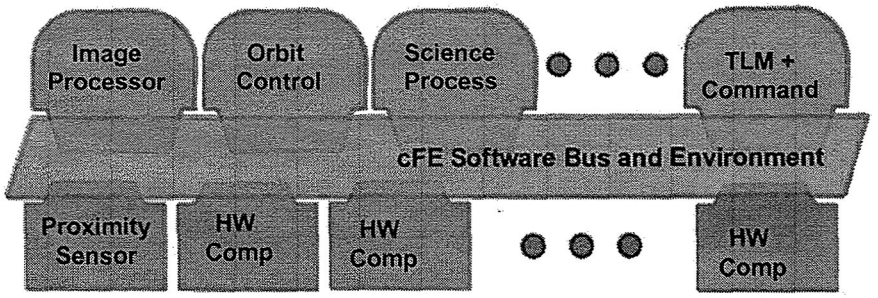
- Components include artifacts, Requirements, Unit tests, build tests, documentation, & code

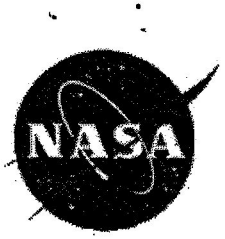


Plug and Play
(Configure and Play)

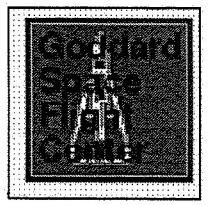
- Common FSW functionality has been abstracted into a catalog of **reusable components and services.**
- **Tested, Certified, Documented**
- **A system is built from:**
  - Core services
  - Reusable components
  - Custom mission specific components
  - Adapted legacy components
- Software components can be configured, built and plugged at any time during development or in flight.
- Software components can be associated with plug in devices or system modes and be loaded as needed







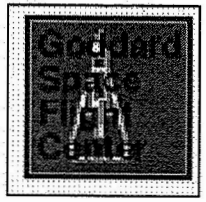
Target Platforms



- **Embedded Systems**
  - PPC/mcp750 running VxWorks 5.5, VxWorks 6.x, Linux
  - PPC/mcp405 running Linux
  - BAE RAD750
- **Desktop Systems**
  - PPC/X86 Macintosh running OS X
  - X86/PC running Linux
  - X86/PC running Cygwin under Windows

Future targets: Coldfire running RTEMS, PPC/mcp750 running RTEMS and LynxOS

Status



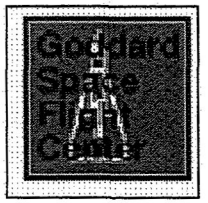
- **2004 multi-CPU/Box prototype demonstrated**
  - Core, generic FSW services and Software components
  - Dynamic application load and startup
  - Dynamic message bus reconfiguration for "Box faults"

- **2005 Version 3.1 cFE, delivered to Autonomy Test Bed (ATB), LRO Nov 16 2005**
  - VxWorks 5.4, Static linking, S-rec file startup
  - For ATB effort, cfE has worked with Vxworks, OSX, and Linux
  - Unit Test Framework (UTF) developed for white box component tests
  - **December CHIPS on orbit demonstration**
    - VxWorks 5.4
    - IP/UDP command and telemetry
    - cFE loaded onto CHIPS providing testbed for automation components

- **2006 Lunar Reconnaissance Orbiter (LRO) configuration baseline cFE 3.3**
  - VxWorks 6.2, Static linking, ELF file startup
  - IP/UDP command and telemetry

- **In process of obtaining open source release of core Flight Executive software**

Conclusion



- By focusing on the spaceflight software domain
- Performing a detailed analysis on heritage mission
- Looking toward future missions
- Allowing for choices

- Implemented a small core framework that
  - Provides the foundation for a component based reuse architecture
  - Reduces cost, schedule and risk to missions
  - Allows for evolution, expansion, and ease of maintenance

Source of Acquisition
NASA Goddard Space Flight Center

A Core Plug and Play Architecture for Reusable Flight Software Systems

Author: Jonathan Wilmot
Affiliation: NASA GSFC
Email: Jonathan.J.Wilmot@NASA.gov

Abstract

The Flight Software Branch, at Goddard Space Flight Center (GSFC), has been working on a run-time approach to facilitate a formal software reuse process. The reuse process is designed to enable rapid development and integration of high-quality software systems and to more accurately predict development costs and schedule. Previous reuse practices have been somewhat successful when the same teams are moved from project to project. But this typically requires taking the software system in an all-or-nothing approach where useful components cannot be easily extracted from the whole. As a result, the system is less flexible and scalable with limited applicability to new projects. This paper will focus on the rationale behind, and implementation of, the run-time executive. This executive is the core for the component-based flight software commonality and reuse process adopted at Goddard.

1. Introduction

Flight software developers are currently faced with some challenging trends. Missions are pushing for more spacecraft autonomy, increasing the mission complexity, and at the same time expecting decreases in the cost and schedule. One approach to reduce cost and schedule is to increase the amount of software reuse. This approach has been used at Goddard with some success but only within the context and funding of a single project limiting the reuse potential. In 2002 the software branch was tasked with two new projects and had some resources to development a common software base. After much heritage system analysis, the concept of a component based reuse model was developed.

To provide an operating basis for the component reuse model, we have developed a small footprint core system executive that supports runtime plug-and-play of software components that conform to the core Application Programmer Interface (API). This allows software developers to select components from a catalog/library and quickly integrate them into new systems achieving project goals of reducing cost and schedule. This system executive is called the core Flight Executive or cFE. The lower case "c" denotes its small footprint.

The core Flight Executive consists of an operating system abstraction layer, a hardware abstraction layer, a set of common system services, and a Publish and Subscribe messaging middleware. All of these functions are designed for run-time plug and play of the catalog/library components. This executive, while facilitating reuse, also adds a great deal of development and operational flexibility in the areas of rapid prototyping, on-orbit software maintenance, redundancy management, and communications.

It is important to note that most of the senior engineers within the Flight Software Branch were involved in the requirements and heritage analysis phases of this development. This involvement was crucial to insure that a wide range of mission experience and lessons learned was captured to develop a truly reusable set of software components and artifacts.

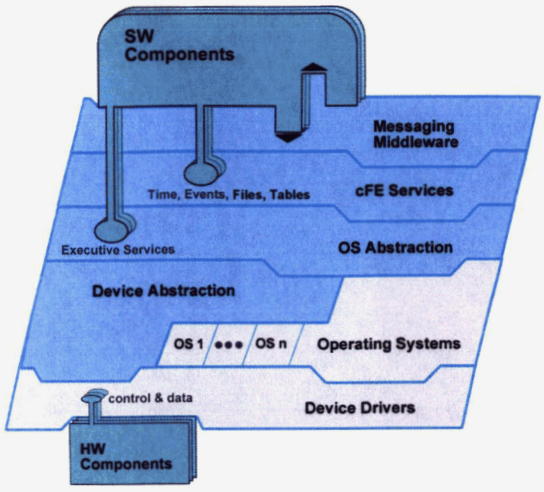


Figure 1
cFE and Abstraction Layers

2. Heritage Analysis

The first phase of the software commonality and reuse model development was to perform a heritage analysis of previous mission software. The intent was to gather the common requirements and best designs from across a set of more than fourteen missions at Goddard ranging from Small Explorers to the Hubble Space Telescope. With the consolidation of flight software engineering within one organization, the reuse team was able to call on many of the original engineers who developed these missions and have them participate in requirements and design reviews. In addition to the mission analysis, some commercial middleware and operation system abstraction products were also reviewed and evaluated using the design goals. Though these were very good products ,they were not able to satisfy the resource utilization goals.

3. Design Goals

Based on the heritage analysis, but with an eye toward future needs, we established several general and specific goals for the flight software reuse architecture. The first goal was to streamline the development process and keep the core system small and simple. The remaining goals fell into the broad categories of performance and the runtime environment and more directly drove the software architecture and its implementation.

3.1 Performance

The performance of space-qualified processing platforms is still relatively low compared to terrestrial systems, with flight processors and support chips a few generations behind. A typical high end flight processing platform may be a 166Mhz PowerPC with 4 megabytes of non-volatile storage and 8 megabytes of RAM. Other spacecraft subsystems or instruments can be even less. These constraints must be considered when developing any flight system.

3.1.1 Software Footprint

The footprint is defined as the memory allocation for the software code. Flight systems store software in the non-volatile memory space and load it into RAM for execution. A target of 512Kbytes to 1Mbytes, depending on the operating system selected, was set as the allocation goal for the core executive. This goal was set to allow the architecture to be used on the smaller subsystems and instruments.

3.1.2 Scaleable Memory Utilization

Memory utilization was defined to be run-time allocation of RAM for code, variables, and configuration data. To maximize scalability, the software architecture needed to meet two goals. The first was to minimize the needs of the core software itself, and the second was to allow the number of applications to vary without making fundamental changes to the architecture. A target of 1 to 2Mbytes was set for the core software, again depending on the operating system selected. The second goal was satisfied by implementing both static and run-time configuration parameters that allowed the number of buffers, files, and tables to scale with the needs of the flight applications.

3.1.3 Processor Utilization

Processor utilization can be simply defined as the percentage of total central processing unit (CPU) time allocated to specific functions over a time period. A goal of 5% utilization over one second was established in line with the heritage systems. In the embedded processor environment, two other parameters must also be considered that cannot be easily specified outside of a specific mission context, that is , peak utilization and application jitter. Those parameters can be positively affected by implementing the core services in ways that minimize blocking calls, semaphore lock times, and other such service optimizations.

3.2 Run-time Environment

The primary goal of the run-time environment is to simplify development and increase on-orbit flexibility. The heritage architecture required that all component interactions be defined at compile time for the entire system. This meant that adding a single telemetry packet to any component would involve compiling and linking the entire system and reloading it during development. On-orbit the user needed to perform the same steps, then difference the executable modules, develop patches for each different area and perform byte writes to those memory locations. This approach was tedious at best and error prone. With this in mind the run-time goals were established.

3.2.1 Component Modularity

In software terms, modularity is a property of programs that measure how well the software parts or modules are decomposed. Modular software typically does one thing and interacts through well-defined interfaces. This property is a goal for reuse. For this effort, another property was added to the classic definition. All components needed to have compile and link modularity, where the files needed to compile the software component are contained within the components directory structure. This goal was established to allow components to be pulled easily from a repository and not require moving files around a complicated directory structure.

3.2.2 Dynamic Loading/Unloading

Dynamic loading is the ability to move a component into the operating environment and start it while the system is running. Unloading is the inverse, stopping the component and removing it from the running system.

3.2.3 Operational Transparency

We use the term operational transparency to refer to the goal of visibility into the operational system state at a given time. This goal was established in response to concerns raised during the heritage analysis. One of the attributes of the heritage design was that system resources and timing could be determined statically by looking at software configuration parameters and structures. With dynamic loading some of these parameters and structures are created at runtime. To be accepted, any new architecture had to support methods to determine resources and timing.

4. Implementation Concepts

Much of the overall implementation can be traced back to the architecture concepts used on one of the first Small explorer missions at Goddard in 1992. Some common reuse concepts, layered software, component application programmer interfaces (API), and the message bus concepts were all implemented even in the early designs. The new architecture formalizes and refines the interfaces, then adds a dynamic property to them. This dynamic property is fundamental to the reuse architecture. Each API resource allocation is tracked and an API is provided to de-allocate resources. For example, buffers, semaphores, pipes, etc, can all be added and removed from the running system by both the application that allocated it and other monitoring applications

4.1. Layered Abstractions

Each layer in the software architecture was design to abstract some variability in the implementation. The core executive software interacts with the layers in a vertical fashion: with the messaging middleware depending on the cFE services, and the cFE services depending on the OS abstraction User components on the other hand, interface horizontally to all three and are restricted to only those three. Operation systems and other interfaces are not exposed to user components, to avoid one of the portability issues in the heritage design.

4.2 Application Programmer Interfaces

Each layer in the architecture has a well-defined set of carefully selected Application Programmer Interfaces. The selection process relied heavily on the heritage analysis to determine which interfaces were required. This kept the number of APIs small and helped meet the design goals of the system.

5. Implementation - The core Flight Executive

The core Flight Executive is the minimal set of component services required for any generic processing element in the flight domain. This core is compiled and linked as a unit generating a single object and several configuration files. The core services are split into 7 logical subsystems that each typically include a library and an external interface application. The 7 subsystems include: Software Bus, Event Handler, OS Abstraction, Executive Services, File Service, Table Management, and Time Management. The library

contains the runtime services and the APIs. The external interface applications implement any required service state machines and contain the high level command and telemetry interfaces for ground system interactions. One of the subsystems, the OS Abstraction, is implemented as a library only. This subsystem may be used without the others as a stand-alone product.

All of the cFE services maintain information for each user component and its tasks in order to provide a way of cleaning up after an application that may have crashed and to provide information to the ground for performance and resource monitoring. This feature is especially useful during development and can be used to restart single components on-orbit.

Software Bus:

The Software Bus (SB) is a Publish and Subscribe messaging middleware that handles both local and distributed and inter-task communications, in a transparent way. This application uses much of the heritage implementation but adds a dynamic middleware interface. The primary abstraction of Software Bus is to provide a mechanism that allows data providers to send packets without knowledge of the data consumers. This allows the one or more subscribers to be on any platform within scope of the bus and not affect the sending application. Software Bus's message-based subscription supports the heritage concepts of one-to-one, one-to-many, and many-to-one routing configurations along with the Poll (non-blocking) and Pend (blocking w/wo timeout) options for message receipt. The quality of service (QoS) concepts of priority and reliability are implemented for all off-processor messages.

Event handler:

Event Handler handles the ground interface, filtering, formatting, sending, and counters for event messages. Event messages are informational text generated by an application in response to commands, software errors, hardware errors, application-initialization, etc. Event messages are sent to alert the Flight Operations team that some significant event on-board has occurred. Event messages may also be sent for debugging application code during development, maintenance, and testing.

OS Abstraction Layer (OSAL):

The OSAL is the API and library abstractions for common operating systems used in development and deployment of flight software systems. To keep it simple, the OSAL implements only the subset of operating system functions as required in heritage analysis and system goals.

The OSAL is a Goddard stand-alone open source project. More information and a list of currently supported operating systems and target platforms can be obtained from,
http://opensource.gsfc.nasa.gov/projects/osal/osal.php

Executive Services:

The Executive Services (ES) subsystem provides system startup, interfaces for run-time control of the cFE and component applications, system logging and interrupt/exception handling. Many of the control features for stopping, starting, suspending, and resuming component applications were not present in the heritage architectures but are required to meet the fundamental design goals.

On system startup the boot firmware copies the cFE, and operating system from non-volatile memory into pre-determined addresses in volatile memory. Control is then transferred to the OS and then to ES which handles the startup of the cFE and the rest of the cFE Applications as indicated in a configuration file. Each application has one main task and may have additional child tasks. The cFE itself is linked and loaded with the RTOS and BSP as a single static executable in non-volatile memory

File Services:

File Service provides access functions for reading and writing standard file headers.

Table Management:

A table is a related set of data values (equivalent to a C structure or array) that can be loaded and dumped as a single unit by the ground. Tables are used in the flight code to give system operators the ability to update constants used by the flight software during spacecraft operation without the need for patching the software. Some tables are also used for dumping infrequently needed status information to the ground on command.

The cFE implements Table Services using a different paradigm than has been used in heritage missions. A Table is considered a shared memory resource. An Application requests the creation of the shared memory from the cFE and the Application must routinely request access and subsequently release access to the Table. In this way, Table Services is able to perform Table updates without the Application being involved.

Time Management:

Time Management handles the ground time interface, intra-processor time distribution and provides the time utilities API for local applications. For intra-processor time distribution, time management provides both server and client interfaces.

7. Putting it All Together- Plug and Play

What the cFE and the build system enables is an environment where individual components can be created, compiled/linked and inserted into a running system. This in turn provides a foundation for a component based reuse catalog and library where much of the component integration is done at run-time, creating true component Plug and Play.

Key to the cFE is the fact that individual cFE applications can be compiled and statically linked separately from the cFE and other cFE applications. Static linking means the global Load Image symbols don't change when the application is bound to the original image.

The results are:

- Faster integration of applications during development
- Facilitation of rapid prototyping and deployment of advanced concepts and algorithms
- Reduced cost, schedule and risk to missions
- Improved flight software maintenance support

8. Conclusion

By focusing on the flight software domain for the common core services and the reuse model, many of the failings of previous reuse attempts have been avoided, thus creating a reuse process that will reduce the cost, risk, and schedule of new mission development. In addition, the implementation has the footprint and run-time performance of the heritage system with only a small startup delay penalty.

A prototype cFE was flown on the CHIPsat mission in December 2005. The first class B mission to use the cFE and library components will launch in late 2008 or early 2009 on the Lunar Reconnaissance Orbiter (LRO).